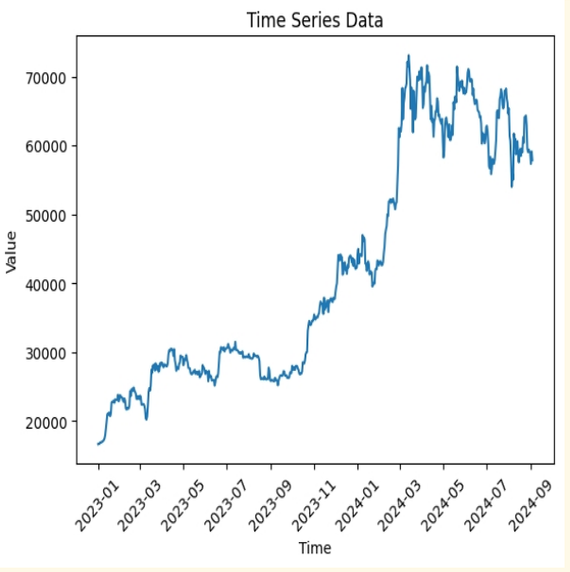
Introduction

Python has quickly emerged as a preferred tool for data analysis due to its simplicity, versatility, and vast community support. With its intuitive syntax and extensive library ecosystem, this elegant programming language allows you to tackle complex problems efficiently. Whether you are building a data-intensive application or working with an experienced data scientist, Python provides a robust platform for exploring, visualizing, and modeling time-dependent data.

Time-series analysis involves examining historical data to uncover patterns, trends, and other valuable insights. It is a crucial step in understanding the behavior of time-dependent data and making predictions for the future.



Python Code

```
import pandas as pd
import numpy as np
import yfinance as yf
import matplotlib.pyplot as plt

df = yf.download("BTC-USD",
                 start="2023-01-01",
                 end="2030-12-31",
                 progress=False)

# Plot the time-series data
plt.plot(data.index, data['Close'])
plt.xlabel('Time')
plt.ylabel('Value')
plt.xticks(rotation = 45)
plt.title('Time Series Data')
plt.show()
```

Python brings a host of benefits to the table regarding Finance (time-series) analysis:

1. It is a **user-friendly** language: Python is known for its simplicity and user-friendliness. Its intuitive syntax makes it easy to learn, even for beginners.
2. It is widely available in the **open-source** world: This means it is freely available to use and is continuously improved and supported by a vibrant community of developers. The open-source nature of Python enables data scientists to access a wealth of resources, tools, and libraries for analyzing time-series data without incurring additional costs.
3. It has **extensive library** support: These libraries, such as pandas, NumPy, Matplotlib, Statsmodels, and Scikit-Learn, provide various functions and tools tailored to the unique challenges of working with time-dependent data. They simplify complex operations, allowing you to focus on extracting meaningful insights rather than reinventing the wheel.

Topic in Our Course

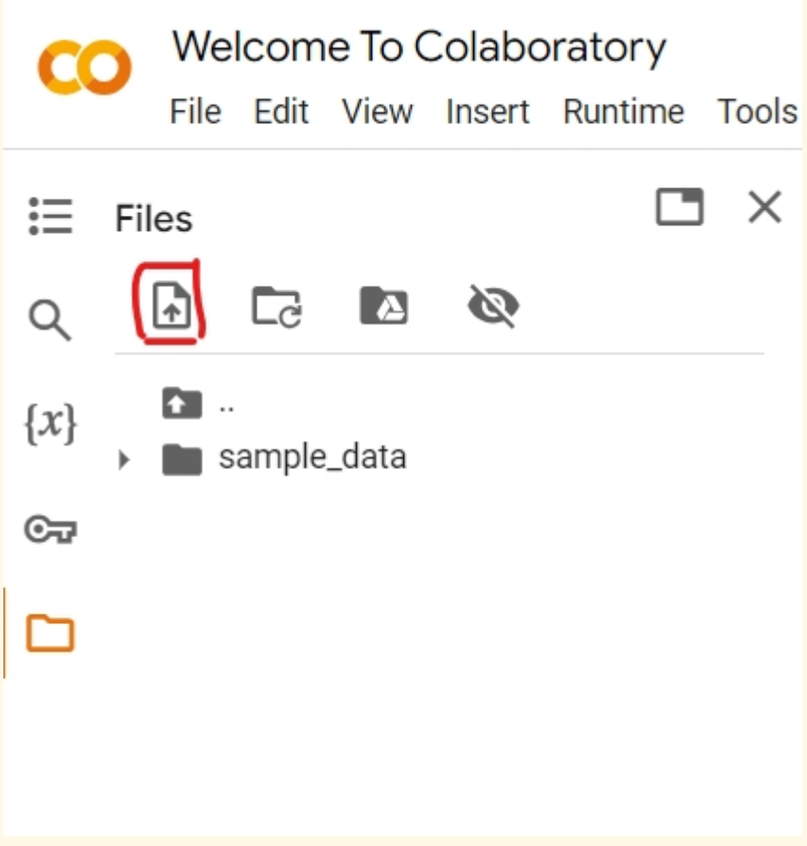
1. **Environment**.
2. **Base Python** : Variable and Data Type, Working with Data, Control Structures and Functions and Modules
3. **Advance Python**: pandas, NumPy, matplotlib, statsmodels, and scikit-learn.
4. **Additional Finance Information**.

IDE for Python

integrated development environment (IDE) refers to a software application that offers computer programmers with extensive software development abilities

Google Colaboratory

Colab for short,is a free Jupyter notebook environment provided by Google where you can use free GPUs and TPUs which can solve all these issues. It contains almost all the modules you need for data science analysis. These tools include but are not limited to Numpy, Scipy, Pandas, etc. Even deep learning frameworks, such as Tensorflow, Keras and Pytorch are also included.



Today's Topics

1. **Simple Python Introduction**
2. **Basic Knowledge**
3. **Variable**
4. **Data Type: Fundamental Types**.

Python History

It was created by Guido van Rossum, and released in 1991. It's great as a first language because it is concise and easy to read. You can use it for web development, to handle big data, machine learning, data science, and software development. So, once you learn the fundamentals, you can grow into the role of your choice!

Python version 1.0 was released in January 1994. The major new features included in this release were the functional programming tools lambda, map, filter and reduce, which Guido Van Rossum never liked. Six and a half years later in October 2000, **Python 2.0** was introduced. Python flourished for another 8 years in the versions 2.x before the next major release as **Python 3.0** (also known as "Python 3000" and "Py3K") was released. Python 3 is not backwards compatible with Python 2.x.

Basic Knowledge

print() Function

The **print()** function in Python is one of the most commonly used built-in functions. It is used to output text, numbers, or other data types to the standard output device (like your screen), or to a file.

```
# print(value1, value2, ..., sep=' ', end='\n', file=sys.stdout, fl
# Default is a newline
print("Hello, World!")
print("Name:", "Alice", "Age:", 25)
print("Orange", "Mango", "Apple")
print("apple", "banana", "cherry", sep=", ")
```

f-strings

It is a way to embed expressions inside string literals, using curly braces `{} `. place expressions (like variables, calculations, function calls, etc.) inside `{}` and Python will replace them with their values.

```
# f"Your text here {expression}"
name = "Alice"
age = 25

print(f"My name is {name} and I am {age} years old.")
portfolio_return = 49.999
```

```
print(f"Expected Portfolio Return: {portfolio_return}")
print(f"Annual Return: {calculate_annual_returns(monthly_returns):.
# :.2f means format the number to 2 decimal places
```

Comments in Python

Comments in Python are used to explain code, make it more readable, or temporarily disable parts of the code during testing or debugging.

```
# Single-Line Comments
# This program displays an address.
print("National Chi-Nan University") #Displays School Name
print("Puli, Nantou")

# Multi-Line Comments
"""
This is a multi-line comment
that explains a block of code.
"""
print("Hello, world!")
```

Indentation is Mandatory

Unlike many other languages that use braces `{}` to define blocks of code, Python uses **whitespace (indentation)** to indicate blocks.

```
# Indentation: **4 space!!!!!!!**
if True:
    print("This is indented.")

from math import sqrt         # block 1
n = input("Maximum Number? ") # block 2
n = int(n)+1                  # block 3
for a in range(1,n):          # block 4
    for b in range(a,n):      # block 4.1 in 4
        c_square = a**2 + b**2
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0): # block 4. 1.1 in 4.1
            print(a, b, c)
```

Case-Sensitive

Python is **case-sensitive** , so `Variable`, `variable`, and `VARIABLE` are all different.

```
name = "Alice"
Name = "Bob"
print(name)  # Outputs: Alice
print(Name)  # Outputs: Bob
```

No Semicolons Needed

```
# Statements:
# single-line statement
print("Hello, Python!")

# use semicolons if you want to write multiple statements in a sing
print("Hello"); print("World")  # Valid, but not commonly used

# multi-line statement
my_list = [
  1, 2, 3,
  4, 5, 6
  ]
```

Keywords Are Reserved

Python has **reserved keywords** that you cannot use as variable names.

```
if, else, for, while, def, class, return, import, try, except, ...
```

Variables Don't Need Declaration

You don't need to declare the type of a variable before using it.

```
x = 10        # Integer
```

```
y = "Hello"   # String
z = 3.14      # Float
```

Variables and Data Types

Variable

A **variable** is a named reference to a value. You can think of it as a **label** for data stored in the computer's memory. we **assign =** value to it.

```
a = 10
b = 15
c = a + b
print("Sum of numbers are", c)

name = "Alice"
```

Rules for Variable Names

1. A variable name can begin with a letter, dollar sign ($),or an underscore (_). It **must not** start with a number.
2. A variable name should describe the kind of information that the variable stores. For example, firstName might be used to store a person's first name. firstName rather than

firstnarne

3. A variable name should not be a **keyword** or reserved words
4. A variable name cannot contain spaces.
5. Uppercase and lowercase characters are distinct (score and Score).

```
age = 25
name = "John"
email_address = "john@example.com"
total_sales = 500.50
first_name = "Mary"
```

```
# Invalid variable names:
2nd_grade = 80    # cannot start with a number
first-name = "John"    # cannot use hyphens
total/sales = 500    # cannot use forward slash
email address = "john@example.com"    # cannot use spaces
if = 10    # cannot use Python keywords as variable names
```

Dynamic Typing

Python is dynamically typed , which means you can reassign variables to different types at any time

```
x = 5          # x is an integer
x = "Hello"    # Now x is a string
x = 3.14       # Now x is a float
```

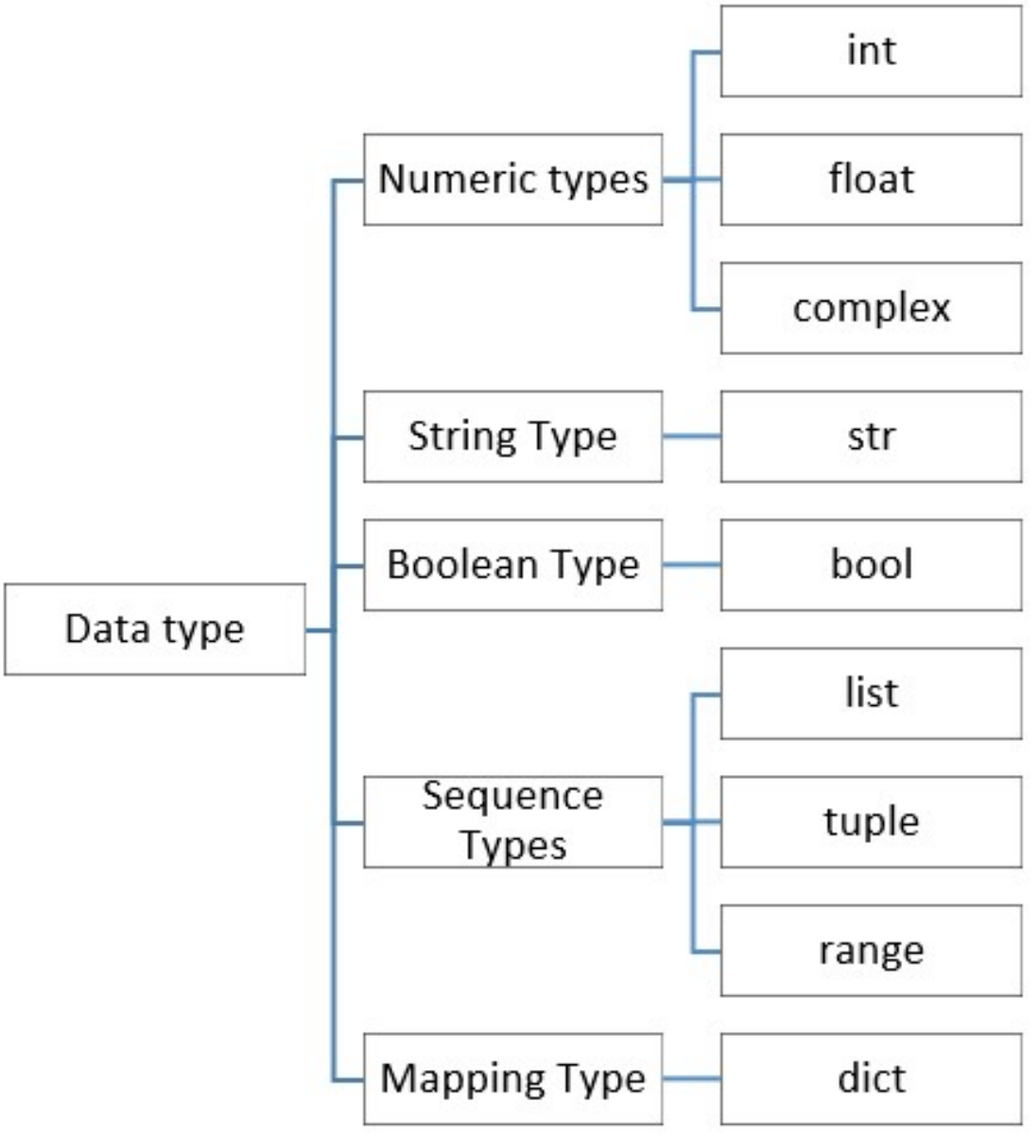
Multiple Assignment

You can assign values to multiple variables in one line. Or assign the **same value** to multiple variables:

```
a, b, c = 1, 2, 3
x = y = z = 0
```

Data Type

In Python, **data types** are used to define the kind of data a variable can hold. Python is a **dynamically typed** language, which means you don't have to explicitly declare the data type of a variable — Python automatically assigns the data type based on the value assigned to the variable.



Fundamental types

Strings

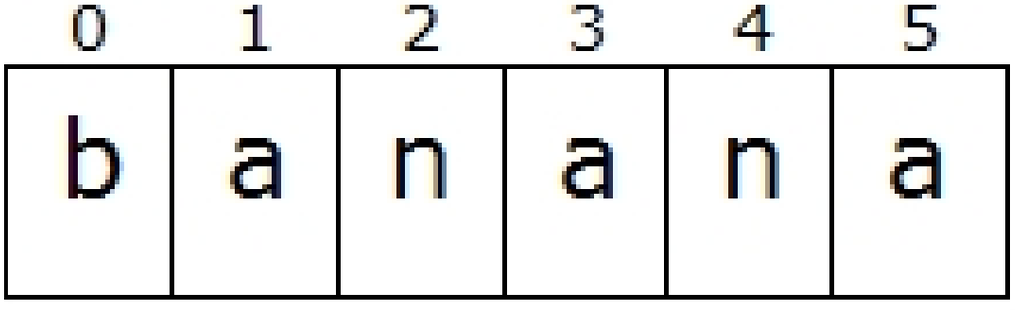
The **`str`** (string) data type in Python is used to represent **textual data** .

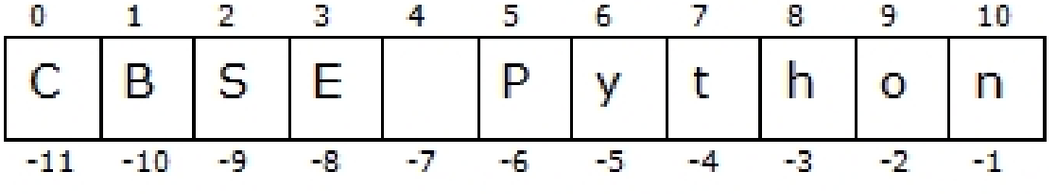
Strings are **immutable sequences** of Unicode characters. You can create

strings using: Single quotes `'...'` or Double quotes `"..."`

```
s1 = 'Harry'
s2 = "Porter"
s3 = '''This is a
multi-line statement.'''
s4 = """Another
multi-line statement."""
```

Each character in a string has an **index** that specifies its **position** in the string





Use a backslash `\` to insert special characters:

Escape sequence	Descriptive
`\n`	New line || ` `	Tab
`\\`	Backslash
`\"`or`\'`	Quotation marks

```
print("Hello\nWorld")  # Prints Hello, then a new line with World
print("He said, \"Hi!\"")  # Prints: He said, "Hi!"
```

```
# slicing
s = "CBSE Python"
print(s[0:7]) # not 7
# CBSE Py

fruit = 'banana'
for letter in fruit:
    print (letter)
```

```
# slicing
first_name = "Harry"
last_name = "Porter"

full_name = first_name + " " + last_name # Concatenation
print(full_name) # Outputs: Harry Porter

# Interesting
"This is a string."
```

```
'This is also a string.'
```

Numeric Types:

Python supports **integers**, **floating point numbers**, and **complex numbers**.

1. **int** : Integers (e.g., `5`, `-3`, `1000`).
2. **float** : Floating-point numbers (e.g., `3.14`, `-0.001`)
3. **complex** : Complex numbers (e.g., `1+2j`, `3j`) .

```
my_integer = 150
my_float = 5.2
sum = my_integer + my_float
print(sum) # Outputs: 155.2

# 4/2 # 2.0
2 * 3.0 # 6.0
3.0 ** 2 # 9.0
```

Boolean Type

Used to represent truth values. Booleans are often the result of comparison or logical operations.

1. **bool** : Either `True` or `False`.

```
x = 10
(x >= 5) & (x <= 11)
```

Dynamic Typing

```
var = 12
print(var) # Outputs: 12

var = "Now I'm a string"
print(var) # Outputs: Now I'm a string
```

Mutable and Immutable Data Types

1. Mutable objects can **change their value** but keep their **id()**, which include *list*, *dict*, *set*..
2. Immutable objects cannot change their value and include **int**, *float*, *bool*, *string*, *tuple*.

Operators Operators

Operators are special symbols or keywords used to perform operations on variables and values. They allow you to manipulate data, perform arithmetic calculations, compare values, make logical decisions, and more. Operators are categorized into several types: *arithmetic*, *comparison*, and *Relational, Logical assignment, and In* operators..

Arithmetic Operators

Used to perform mathematical operations.

```
# Arithmetic Operation
sum = 5 + 3
difference = 5 - 3
product = 5 * 3
quotient = 5 / 3
remainder = 5 % 3
quotientA = 5 // 3
power = 5 ** 3

10 / 3

10.0 / 3.0

9 // 3

10 // 3

11//3
```

Comparison Operators

Used to compare two values. Returns a **boolean** (`True` or `False`)

```
# Relational Operators - True or False
5 == 3 # Equal to
5 != 3 # Not equal to
5 > 3  # Greater than
5 < 3  # Less than
5 >= 3 # Greater than or equal to
5 <= 3 # Less than or equal to
```

Assignment Operators

Used to assign values to variables

```
# Assignment Operators
x = 5
x += 5 #  x = x + 5
x -= 5 #  x = x - 5
x *= 5 #  x = x * 5
x /= 5 #  x = x / 5
x %= 5 #  x = x % 5
x //= 5 #  x = x // 5
x **= 5 #  x = x ** 5
```

Logical Operators

Used to combine conditional statements.

```
# Logical Operator
x = 6
x < 5 and x < 10 # Returns True if both statements are true
# False
x < 5 or x < 10 # Returns True if at least one statement is true
# True
not(x < 5 and x < 10) # Reverses the result
# True
```

Membership Operators

Used to test whether a value is present in a sequence (like string, list, tuple, etc.)..

```
# in Operator
number = 33
number in (10,4,33,99) Returns True if value is found
# True

# not in operator
'b' not in 'apple' # Returns True if value is NOT found
```